Chapter 1

Dynamically Extending the Eclipse ICE UI

This tutorial will show you how to create custom, dynamic UI extensions to Eclipse ICE.

What you will need for this tutorial:

- Experience creating Eclipse plugins
- Experience writing UI code with SWT
- Experience creating an ICE Item

1.1 Introduction

ICE makes some educated guesses based on the type of your components and information that it can glean from your data to figure out the best way that it can generate the the UI. However, after you create your first set of Items, you might find yourself wondering if you can change the way that ICE auto-generates the UI to better fit your needs. ICE lets you do this by setting the Context of your Form and Components with the setContext() operation. Setting the context with a string that is unique to your project will let ICE look up UI extensions that you create and publish through the Eclipse 4 framework.

There are several important things to consider before you start extending the UI. First, how much work do you want to do? Some of the UI constructs in ICE are quick to change, such as EntryComposites for showing Entries, but others, like GeometryPage and MeshPage, could require significant work because of the evel graphics involved.

This tutorial will show you how to change two pieces of ICEs UI: The page for showing GeometryComponents and an EntryComposite. We will only show you how to change the GeometryPage, not how to actually generate new 3D

graphics. The source code for this tutorial is in the ICE repo in a project called org.eclipse.ice.demo.

1.2 Create an ICE Item Project

Use the ICE Item Project Generation Wizard to create a new Item with a eometryComponent and a DataComponent with one Entry in the Form. You can copy the following code into your setupForm() operation:

```
@Override
    public void setupForm() {
        form = new Form();

        ioService = getIOService();

        // Create a geometry component
        GeometryComponent geomComp = new GeometryComponent();
        geomComp.setName("Geometry");
        geomComp.setDescription("A geometry");
        geomComp.setContext("demo-geometry");
        geomComp.setGeometry(
                new ShapeController(new ShapeMesh(), new AbstractView()));

        // Create a data component
        DataComponent dataComp = new DataComponent();
        dataComp.setName("Data");
        dataComp.setDescription("Some Data");
        dataComp.setContext("demo");
        // Need to set the id since geomComp is number 1
        dataComp.setId(2);

        // Create an Entry for the data component
        IEntry entry = new StringEntry();
        entry.setName("Data Entry");
        entry.setDescription("An Entry with Important Data");
        entry.setContext("demo-entry");
        // Add the Entry to the data component
        dataComp.addEntry(entry);

        // Add both components to the Form, showing the data component
            first.
        form.addComponent(dataComp);
        form.addComponent(geomComp);

        // Set the context on the Form
        form.setContext("demo");

        return;
    }
```

Note that your import packages lines should look this the following:

```
import org.eclipse.core.resources.IFile;
import org.eclipse.core.resources.IProject;
import org.eclipse.core.runtime.CoreException;
import org.eclipse.eavp.viz.service.modeling.AbstractView;
import org.eclipse.eavp.viz.service.modeling.ShapeController;
import org.eclipse.eavp.viz.service.modeling.ShapeMesh;
import org.eclipse.ice.datastructures.entry.IEntry;
import org.eclipse.ice.datastructures.entry.StringEntry;
import org.eclipse.ice.datastructures.form.DataComponent;
import org.eclipse.ice.datastructures.form.Form;
import org.eclipse.ice.datastructures.form.FormStatus;
import org.eclipse.ice.datastructures.form.GeometryComponent;
import org.eclipse.ice.io.serializable.IIOService;
import org.eclipse.ice.io.serializable.IReader;
import org.eclipse.ice.io.serializable.IWriter;
import org.eclipse.ice.item.model.Model;
```

You may need to add some of these packages to your Manifest file.

If you have not created an Item in ICE before, please see ¡the ICE Item Generation tutorial¿ to do this.

Add this plugin to the launch configuration for your system, launch ICE and make sure that you can successfully create a DemoModel Item. You should have see a Form with two pages as in figures 1.1 and 1.2.

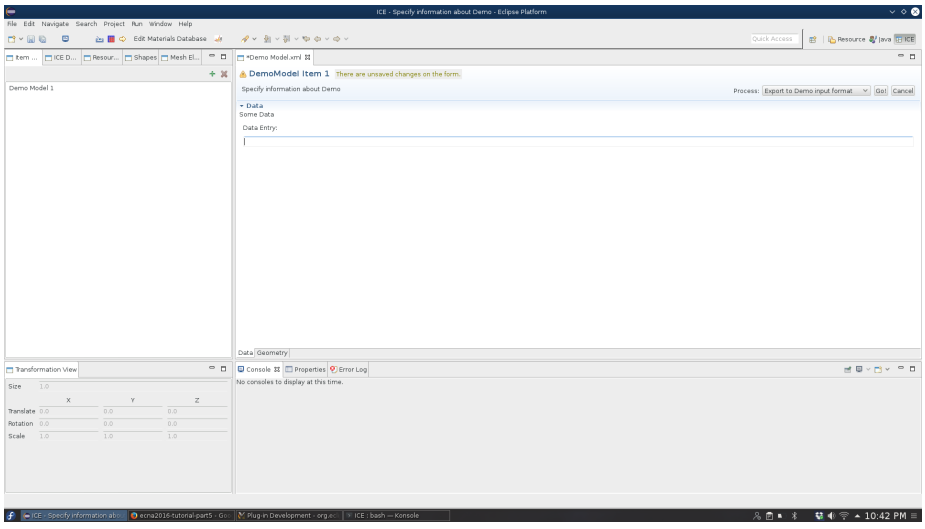

Figure 1.1: Default Entry Composite on a Form

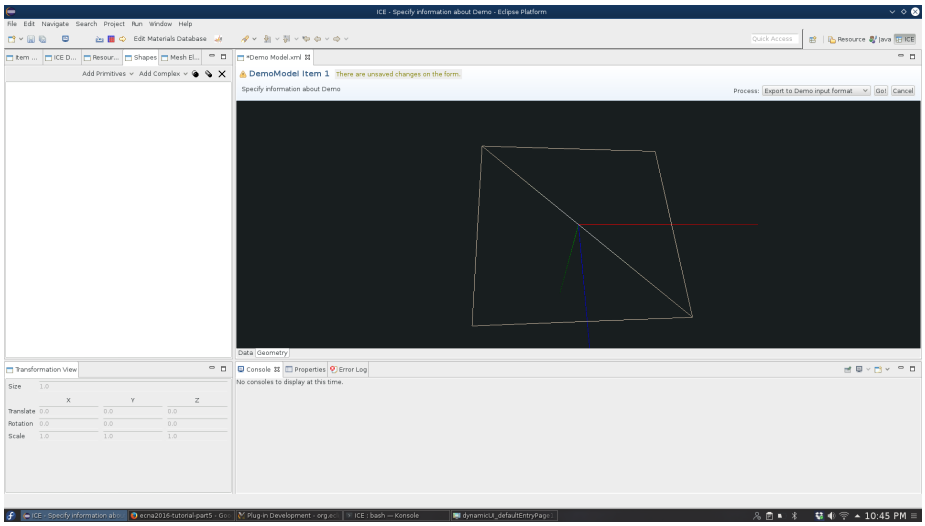

Figure 1.2: Default Geometry Page

The rest of the tutorial will look at replacing the routine that draws the Entry on the first page and the entire Geometry Page with your own custom page. You wont have to make any additional changes to your Item (assuming it works as shown above). Note that in the code above we need to use different context names for the Entry and the GeometryComponent so that the context can uniquely identify the routines that draw each.

1.3 Create an IPageProvider and IPageFactory

We will replace the Geometry Page first. Create a new package.This requires two pieces to properly locate your Page, through your IPageFactory, and to draw your Page, through you IPageProvider.

Start by adding the following packages to your import packages block in your Manifest file:

- org.eclipse.ice.client.widgets.providers
- org.eclipse.ice.client.widgets.providers.Default
- org.eclipse.ui.forms
- org.eclipse.ui.forms.editor
- org.eclipse.ui.forms.widgets
- org.eclipse.e4.core.contexts
- org.eclipse.e4.core.di

- org.eclipse.e4.ui.model.application
- org.eclipse.e4.ui.model.application.ui

Create a class that implements IPageProvider and copy the following code into it:

```
public class DemoGeometryPageProvider implements IPageProvider {

    @Override
    public String getName() {
        // TODO Auto-generated method stub
        return "demo";
    }

    @Override
    public ArrayList<IFormPage> getPages(FormEditor formEditor,
            ArrayList<Component> components) {

        ArrayList<IFormPage> pages = new ArrayList<IFormPage>();
        // Get the GeometryComponent and create the GeometryPage.
        if (!(components.isEmpty())) {
            GeometryComponent geometryComponent = (GeometryComponent)
                (components
                   .get(0));

            if (geometryComponent != null) {

                // Make the GeometryPage
                DemoGeometryPage geometryPage = new
                    DemoGeometryPage(formEditor,
                       "GPid", geometryComponent.getName());

                // No need to set the geometry component for the demo, but
                // something like would be necessary in a real
                    application.
                // geometryPage.setGeometry(geometryComponent);

                // Add the page
                pages.add(geometryPage);
            }

        }

        return pages;
    }

}
```

This class will provide your page to the platform. We need a second class

that actually draws the content on the Geometry page. We will create a third class here, for convenience, that inherits from ICEs ICEFormPage base class to act as the GeometryPage. We dont actually need to show the content of the Geometry component for now because we just want to show that we can change the page. So, create your class and copy the following code into it to just show a label in place of the original 3D editor:

```
public class DemoGeometryPage extends ICEFormPage {

    public DemoGeometryPage(FormEditor editor, String id, String title) {
        super(editor, id, title);
        // TODO Auto-generated constructor stub
    }

    /**
     * <p>
     * Provides the page with the geometryApplication's information to
         display
     * geometry.
     * </p>
     *
     * @param managedForm
     *           the managed form that handles the page
     */
    @Override
    public void createFormContent(IManagedForm managedForm) {

        // Local Declarations
        final ScrolledForm form = managedForm.getForm();
        GridLayout layout = new GridLayout();

        // Setup the layout and layout data
        layout.numColumns = 1;
        form.getBody().setLayoutData(
                new GridData(SWT.FILL, SWT.FILL, true, true, 1, 1));
        form.getBody().setLayout(new FillLayout());

        // Just create some text and say hello
        Label geometryText = new Label(form.getBody(), SWT.FLAT);
        geometryText.setText("Draw something based on the geometry.");

        return;

    }

}
```

1.4 Create a new IEntryComposite

Individual Entrys that ICE draws can be extended in the same way. First, create a subclass of AbstractEntryComposite that implements render and copy the following code into it:

```
public class DemoEntryComposite extends AbstractEntryComposite {

    public DemoEntryComposite(Composite parent, IEntry refEntry, int
        style) {
        super(parent, refEntry, style);
    }

    @Override
    public void render() {

        Button button = new Button(this, SWT.PUSH);
        button.setText("My button");

        Label label = new Label(this, SWT.FLAT);
        label.setText(super.getEntry().getValue() + " in new Entry
            widget.");
        setLayout(new FillLayout());
        this.layout();

        return;
    }

}
```

Next, create a provider to publish your Entry Composite to the platform and copy the following code into it:

```
public class DemoEntryCompositeProvider implements
    IEntryCompositeProvider {

    @Override
    public String getName() {
        // TODO Auto-generated method stub
        return "demo-entry";
    }

    @Override
    public IEntryComposite getEntryComposite(Composite parent, IEntry
        entry,
           int style, FormToolkit toolKit) {
        // TODO Auto-generated method stub
        return new DemoEntryComposite(parent, entry, style);
    }
```

```
}
```

1.5 Publishing through the e4 extension point

The most common way to publish these extensions to the platform is to create an extension at the org.eclipse.e4.workbench.model extension point. You can do this by creating an executable processor. Create a new class and add the following code:

```
public class DemoWidgetsProcessor {

   /**
    * This operation executes the instructions required to register the
        demo
    * widgets with the e4 workbench.
    *
    * @param context
    *           The e4 context
    * @param app
    *           the model application
    */
   @Execute
   public void execute(IEclipseContext context, MApplication app) {

      // Add the geometry provider
      IPageProvider provider = ContextInjectionFactory
            .make(DemoGeometryPageProvider.class, context);
      context.set("demo-geometry", provider);

      // Add the EntryComposite provider
      IEntryCompositeProvider compProvider = ContextInjectionFactory
            .make(DemoEntryCompositeProvider.class, context);
      context.set("demo-entry", compProvider);

   }

}
```

Next, either create an extension point graphically or add the following code to your plugin.xml file:

```
   <extension
         id="org.eclipse.ice.demo.ui.processor"
         name="ICE Demo UI Processor"
```

```
      point="org.eclipse.e4.workbench.model">
   <processor
         apply="always"
         beforefragment="true"
         class="org.eclipse.ice.demo.ui.DemoWidgetsProcessor">
   </processor>
</extension>
```

Your extensions will now be available in the workbench. Your entry composite is should look like fig. 1.3.

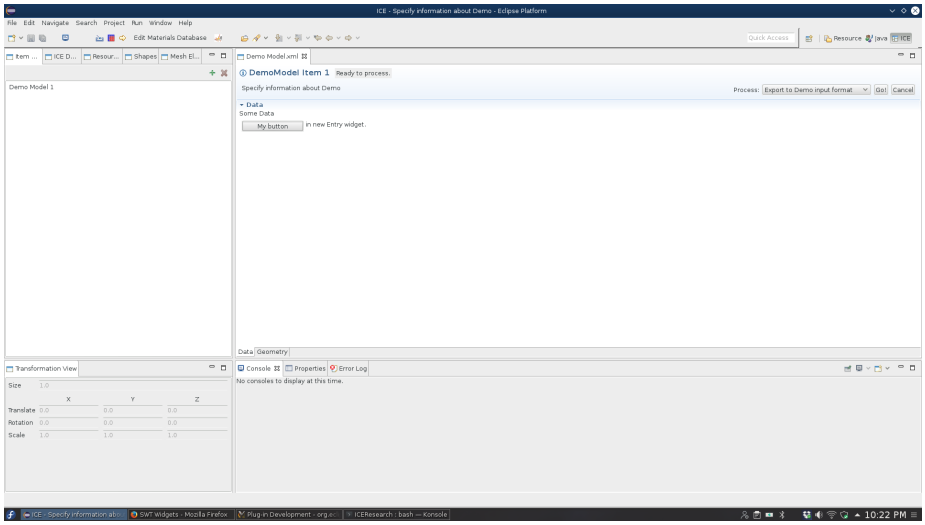

Figure 1.3: Updated Entry Widget

You geometry component should look like fig. 1.4.

1.6 Publishing through Context Functions

Make sure to check Activate this plug-in when one of its classes loaded. in your MANIFEST.mf file.

You can alternatively publish your new extensions to the OSGI framework so that they can be dynamically injected into the ICEs UI code. These extensions are registered as standalone services that are dynamically located at runtime based on the context name that you specified in your data structures. The benefit of this method over the extension point is that it can be used to dynamically update based on the *present* context of the UI, not the initial context. This means that in addition to the type of data structure involved the UI can be tailored based on the particular area of the workbench where it would be drawn and the current runtime state.

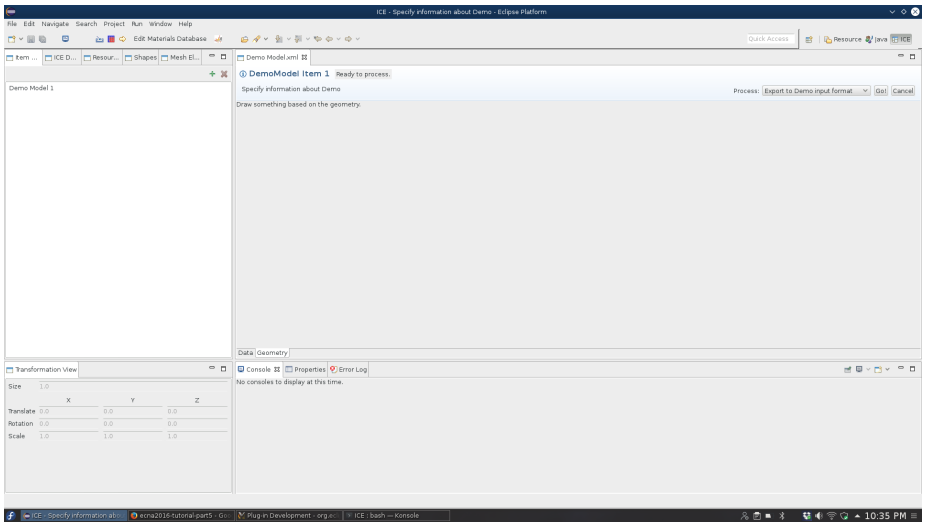

Figure 1.4: The updated Geometry page.

Create a new context function for your IEntryComposite. You should create a subclass of ContextFunction with the following code:

```
public class DemoEntryCompositeContextFunction extends ContextFunction {

    @Override
    public Object compute(IEclipseContext context, String contextKey) {
        IEntryCompositeProvider provider = ContextInjectionFactory
                .make(DemoEntryCompositeProvider.class, context);
        // add the new object to the application context
        MApplication application = context.get(MApplication.class);
        IEclipseContext ctx = application.getContext();
        ctx.set(IEntryCompositeProvider.class, provider);
        return provider;
    }

}
```

Create an OSGI component with the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
name="org.eclipse.ice.demo.entry"> <implementation
   class="org.ecli pse.ice.demo.ui.DemoEntryCompositeContextFunction"/>
   <property name="service.context.key" type="String"
       value="demo-entry"/>
   <service> <provide
   interface="org.eclipse.e4.core.contexts.IContextFunction"/> </service>
```

```
</scr:component>
```

Now, create a context function for your Geometry Page. You should create a subclass of ContextFunction with the following code:

```
public class DemoGeometryPageContextFunction extends ContextFunction {

    @Override
    public Object compute(IEclipseContext context, String contextKey) {
        IPageProvider provider = ContextInjectionFactory
                .make(DemoGeometryPageProvider.class, context);
        // add the new object to the application context
        MApplication application = context.get(MApplication.class);
        IEclipseContext ctx = application.getContext();
        ctx.set(IPageProvider.class, provider);
        return provider;
    }

}
```

and an OSGI component with the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
name="org.eclipse.ice.demo.geometry.page"> <implementation
class="org.eclipse.ice.demo.ui.DemoGeometryPageContextFunction"/>
    <property
name="service.context.key" type="String" value="demo-geometry"/>
    <service>
     <provide
         interface="org.eclipse.e4.core.contexts.IContextFunction"/>
  </service>
</scr:component>
```

1.7 Replacing the whole Page Factory

If you want to replace the way that all pages in ICE are drawn, you can replace the entire Page Factory. Create a new class in your package that inherits from DefaultPageFactory, which will save you some work over implementing an entire new factory from scratch. See fig. 1.5.

Copy the following code into your new subclass:

```
public class DemoGeometryPageFactory extends DefaultPageFactory {

    @Override
    public ArrayList<IFormPage> getGeometryComponentPages(FormEditor
        editor,
```

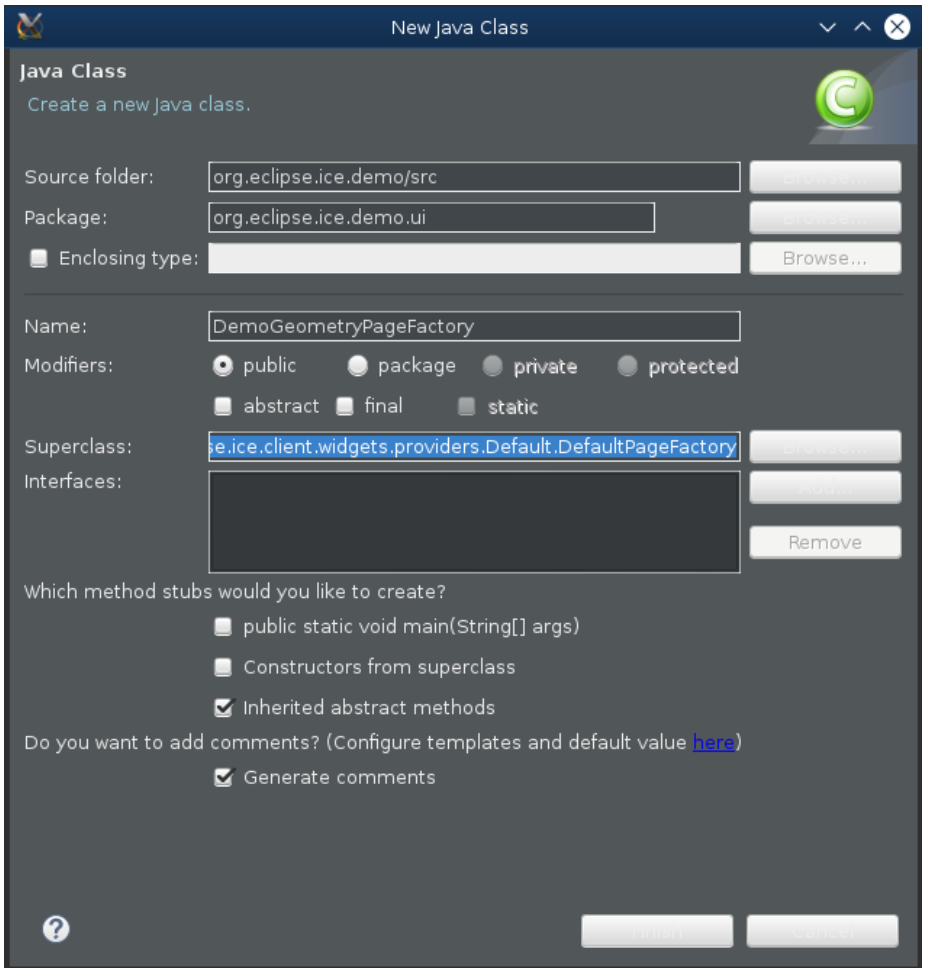

Figure 1.5: Generating the Page Factory class.

```
            ArrayList<Component> components) {
        DemoGeometryPageProvider pageProvider = new
             DemoGeometryPageProvider();
        return pageProvider.getPages(editor, components);
    }
}
```

This will now build on your previous DemoGeometryPageProvider to provide it to the framework through a factory. You can override other operations to provide access to other custom pages. This method may be more efficient than creating separate page extension for each page type. To publish this to the platform, create a Context Function with the following code:

```
public class DemoGeometryPageFactoryContextFunction extends
    ContextFunction {

    @Override
    public Object compute(IEclipseContext context, String contextKey) {
        IPageFactory factory = ContextInjectionFactory
                .make(DemoGeometryPageFactory.class, context);
        // add the new object to the application context
        MApplication application = context.get(MApplication.class);
        IEclipseContext ctx = application.getContext();
        ctx.set(IPageFactory.class, factory);
        return factory;
    }

}
```

Your OSGI component should be published as such:

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
name="org.eclipse.ice.demo.factory"> <implementation
class="org.eclipse.ice.demo.ui.DemoGeometryPageFactoryContextFunction"/>
<property name="service.context.key" type="String" value="demo"/>
    <service>
     <provide
         interface="org.eclipse.e4.core.contexts.IContextFunction"/>
  </service>
</scr:component>
```

and your MANIFEST.mf file should contain:

```
Service-Component: OSGI-INF/*.xml
```